Object-Oriented Design And Programming

Cecilia M. Castillo
Elizabeth B. Flanagan
Nancy M. Wilkinson

There is growing interest and need within AT&T to accelerate use of object-oriented technology in application development. The promise of this technology is for smaller, more maintainable, and more easily extensible software products than can be produced with traditional approaches. However, the technology requires that software developers view the world and approach problems in a fundamentally different way. In this article, we discuss object-oriented design and object-oriented programming and their potential benefits. We also describe the status of object-oriented technology within AT&T and share experiences of some customers we have worked with and supported during the last five years.

Object-Oriented Design and Programming

Object-oriented design, according to Booch,[1] is the process of decomposing a problem into parts that represent classes or objects from the problem domain. Object-oriented design views a problem and solution as a collection of objects that cooperate to achieve some functionality. It is much like organizing a set of people, with their own individual knowledge, talents and responsibilities, to perform a task. Languages that support object-oriented programming, such as C++, support these classes and collaborations.

Object-oriented programming and low level object-oriented design are so intertwined that it is nearly impossible to draw sharp distinctions between them. Object-oriented programming refers to the language-dependent portion of object-oriented design where the effective use of a particular language mechanism is important. In our experience, the end products of a successful design process are well-defined C++ classes that include objects specific to the problem domain and the design. In this article, when we say object-oriented design, we mean object-oriented design and programming.

Object-Oriented Technology in AT&T

At the core of the object-oriented movement within AT&T is the C++ language.[2] It is based on work begun at AT&T Bell Laboratories in the early 1980's by Bjarne Stroustrup, who developed an extension of the C programming language called "C with Classes." Renamed C++[3] in 1983, the language has opened a pathway for migration from C-based programming techniques to object-oriented design and development.[4] C++, now in its third year of becoming an ANSI standardized process, has become a stable, full-scale production language.[5] Environments supporting object-oriented development are constantly improving and being ported to increasingly diverse platforms.

Object-oriented technology has its share of emerging, though immature, methods and tools for analysis and design. Design tools supporting responsibility-driven design methods, such as those of Booch, Wirfs-Brock,[6] or preliminary design activities, such as Class, Responsibility, Collaborator (CRC) cards,[7] are now emerging. Some CASE vendors are migrating their structured analysis and design tools to data-driven methods, such as Schlaer and Mellor.[8] However, since there is insufficient information available from real project experiences to validate the claims about many of the tools and methods, we will not present any specific evaluations but instead focus on the benefits of object-oriented design as seen in AT&T and discuss specific project experiences relating to these benefits.

While only a few projects are cited specifically, our comments are based not only on our own personal experience of developing the language, standard component libraries, and tools but also on our extensive internal and external customer base developed during the past five years. We have maintained close relationships with projects, continuously increasing our understanding of the process, benefits, trends, pitfalls, and supporting technologies.

Benefits of Object-Oriented Design

Object-oriented design decomposes a system into entities, called *objects*, each of which captures its own information and behaves in a certain way. Each object knows how to play its role in the operation of the system. Objects with similar information and behavior are grouped into classes; thus, an object is an instance of a class. Object-oriented design has important benefits over algorithmic, or procedural, approaches. The object-oriented view:

- is better at organizing inherently complex software systems,
- reduces development effort through reuse,
- yields systems that are more resilient to change,
- yields systems that are better able to evolve.

We will take a close look at each benefit, providing examples from actual experience. Code examples, while from real projects and reflecting real code, are sometimes presented in a slightly altered form to protect proprietary information.

Better Organization of Inherent Complexity. Today's software systems address problems that are difficult and complex. Such applications are usually too complex for any single individual to comprehend. While no technology will make these difficult problems easy to solve, the object-oriented approach can make them more manageable.

Single vocabulary. Object-oriented design allows software developers to more directly model the problem domain by using objects and classes as the basis for a single vocabulary throughout the software process. The classes in the domain are the same as those in the design and the implementation. This makes the solution more intuitively comprehensible, and is a huge step toward managing complexity. For example, one project team built an environment for analog circuit simulation, where C++ was also used as the hardware description language. Class names in this environment include *Circuit, IdealIsink, CurrentMirror, Node,* and *Device.* Project members claim that using object-oriented design and C++ permitted them to easily describe circuits as they understood them, using C++ classes, and provided languange support for these classes.

Higher level of abstraction. The object-oriented approach manages complexity by abstracting knowledge and encapsulating it within the objects of the design. *Encapsulation* means that data and operations upon that data are grouped into single objects. A technique called *information hiding* allows some of these data and operations to be private to the object, meaning no other object can access them or need know anything about them. Only the public side of an object need be known to understand what the system does. The abstraction or *chunking* quality that objects provide increases the overall comprehensibility of a a system.

For example, a library of C++ OPEN LOOK® Widgets hides the details of X and OPEN LOOK within C++ classes. (OPEN LOOK is a registered trademark of UNIX Systems Laboratories, Inc.) The following example illustrates a pop-up window that displays a question and asks for a yes or no response. It then causes an action to take place based on the reply it receives. To use and understand this class, a programmer need understand only the public interface:

```
class YesNoPopup {
public:
   YesNoPopup(const char* message,
      yesAction, noAction);
   void getYesNo();
   void doYes();
   void doNo();
private:
// X and OPEN LOOK details
}
```

Reuse Reduces Development Effort. Reusing classes that have been written, tested and maintained by others cuts development, testing and maintenance time. Reuse of locally constructed classes intended for a particular project or family of projects also provides enormous savings. Project teams have realized code production savings amounting to 30 to 50 percent of the total lines of code through internal and/or class library reuse.

Many projects have taken advantage of the class libraries distributed with *cfront* within AT&T and now

sold externally as the USL C++ Standard Components by UNIX Systems Laboratories Inc. One such project produced a computer-aided design tool for designing assemblies of mechanical parts that took advantage of general purpose *Lists, Maps, Strings, Graphs, Bits, Objection, Blocks* and *Pool* classes. Project members estimate that reusing these libraries saved 12 to 18 staff-months that would have been needed to develop similar but less widely reusable classes. They also claim that these libraries helped increase the overall quality of their product.

General purpose class libraries such as *Strings, Lists* and *Maps* represent the initial effort at understanding and generating reusable class libraries. The lessons learned from the production of general purpose libraries are now being applied to the development of more specific application domain class libraries and to the construction of frameworks.

More Extensible and Maintainable Systems. Maintenance, including product enhancements and repairs, traditionally commands about 65% of the cost of any product life-cycle. Improving the ease of such a significant portion of the product development offers obvious advantages.

Changing systems. Local control, encapsulation, and information hiding make systems more resilient to change. By helping limit the number of potential interactions of different parts of the software, they ensure that changes to the implementation of a class can be made with little impact on the rest of the system. For example, bug fixes are more localized and will not have unintended effects in other parts of the system. One project team, which wrote a second version of a C-based network support system for tracking maintenance information in C++, reported that the number of different code changes needed to respond to a single reported bug decreased dramatically with their C++ implementation.

Another benefit of encapsulation and information hiding is that whole subsystems can be moved in and out of an application without affecting the rest of the system. A common approach in object-oriented development applications that use relational database management systems (DBMS) is to encapsulate calls to the database in interface classes. This will allow relatively easy migration to a different DBMS should this become a customer requirement during the initial development cycle or in later phases. For example, the following example shows a class that encapsulates a database table construction could be part of a DBMS interface:

```
class DBTable {
// The member functions are all implemented
// using the mechanisms of a particular
// database management system.
public:
   DBTable();   // constructor
   void insert(); // insert data
   void remove(); // delete data
   void update(); // change data
   void alter();  // modify structure of table)
}
```

Because of performance problems, a fairly large operations system project needed to change its underlying DBMS midway through development. About four weeks was spent rewriting the implementation of the interface classes such as the example above. Virtually no time was spent altering any other part of the system because no other part needed to understand this internal implementation change protected by encapsulation.

Extending the system. A subclass/superclass relationship between classes exists when one 'is a kind of' the other. For example, an object-oriented work management system presently under construction has classes *WorkItem, RepairWorkItem* and *InstallWorkItem.* Since RepairWorkItem and InstallWorkItem 'are kinds of' WorkItem, they are subclasses of it. RepairWorkItem and InstallWorkItem inherit state and behavior from WorkItem, their superclass.

The property of object-oriented design that allows for easy extensibility is sometimes called *polymorphism.* Polymorphism is the notion that any request sent to a superclass object can be automatically carried out by any object of one of its subclasses. Objects of the subclass will respond to that request in a way appropriate for that object. For example, a 'close' message can be sent to any object that is a kind of WorkItem and that object will close itself correctly.

Polymorphism allows the developer to extend the set of classes in a system simply by adding a definition of a new subclass. For example, a new kind of WorkItem called MaintainenceWorkItem can be used in the work management system simply by defining such a class. Then, all requests handled by WorkItems will be handled appropriately by MaintainenceWorkItem objects with no change to the rest of the system.

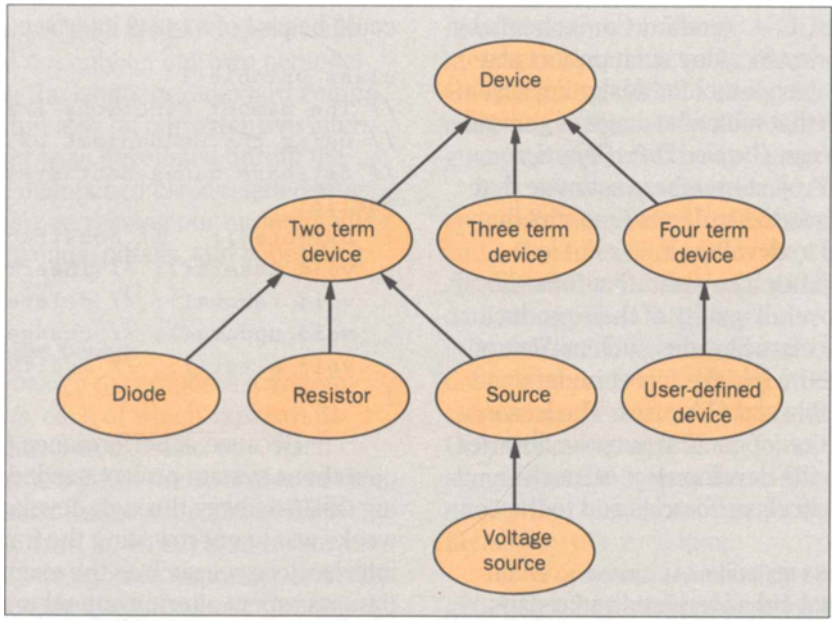

Figure 1. This example of a Device class hierarchy shows how new devices can be built by creating subclasses of existing devices and defining the physical law of the new device.

An effective use of polymorphism was made by a project that built the environment for analog circuit simulation. Anticipating the need for different kinds of primitive devices as parts of circuits, they constructed a *Device* interface class to specify the common features of devices such as scale and temperature. Classes such as *TwoTermDevice, ThreeTermDevice* and *FourTermDevice* provide features common to devices with two, three, and four terminals and are implemented as subclasses of Device. Primitive devices are implemented as further specializations in the Device class hierarchy as shown in Figure 1. New devices, even user-defined ones, can be built by creating subclasses of existing devices and defining the physical law of the device in the "eval" operation. No other change in the simulation code is necessary.

OPEN LOOK Widget Library Example

Now we will take a closer look at the OPEN LOOK Widgets example referred to in the previous section. This example will illustrate the benefits of object-oriented design and programming. The class designs presented here are from a class library originally developed to meet specific project needs. It was subsequently used by several related projects. This library of C++ classes was constructed to encapsulate and "objectize" the OPEN LOOK Widgets to fit them into the object-oriented paradigm more seamlessly. There are about fifty classes and fifteen thousand lines of code.

This library's author was working on a C programming environment for developing, browsing, and maintaining C program code. Based on his experience and working relationships with other product developers, he anticipated a further use for such a library. So he spent time designing, developing, and testing a generic library, which freed other projects from the tedium of having to program user interfaces.

New users can benefit from reuse by deriving new classes from the library base classes. The use of virtual functions allows for easy extension through polymorphism. New C++ Widget classes can be created by programmers by combining existing widgets to create useful composite widgets. For example, a *Caption* and a *TextEntry* widgets were combined to create a *LabelledTextEntry*.

If we look at part of the Widget inheritance tree shown in Figure 2, we see form, text field, menu and popup widget leaf classes. All of these classes are the result of several layers of inheritance.

This class library is currently used by many related projects. These products focus on tool construction; and this library helps to implement tools with the same "look and feel". The library has significantly reduced the complexity of programming in X-Windows since C++ programmers only need to understand the public behavior

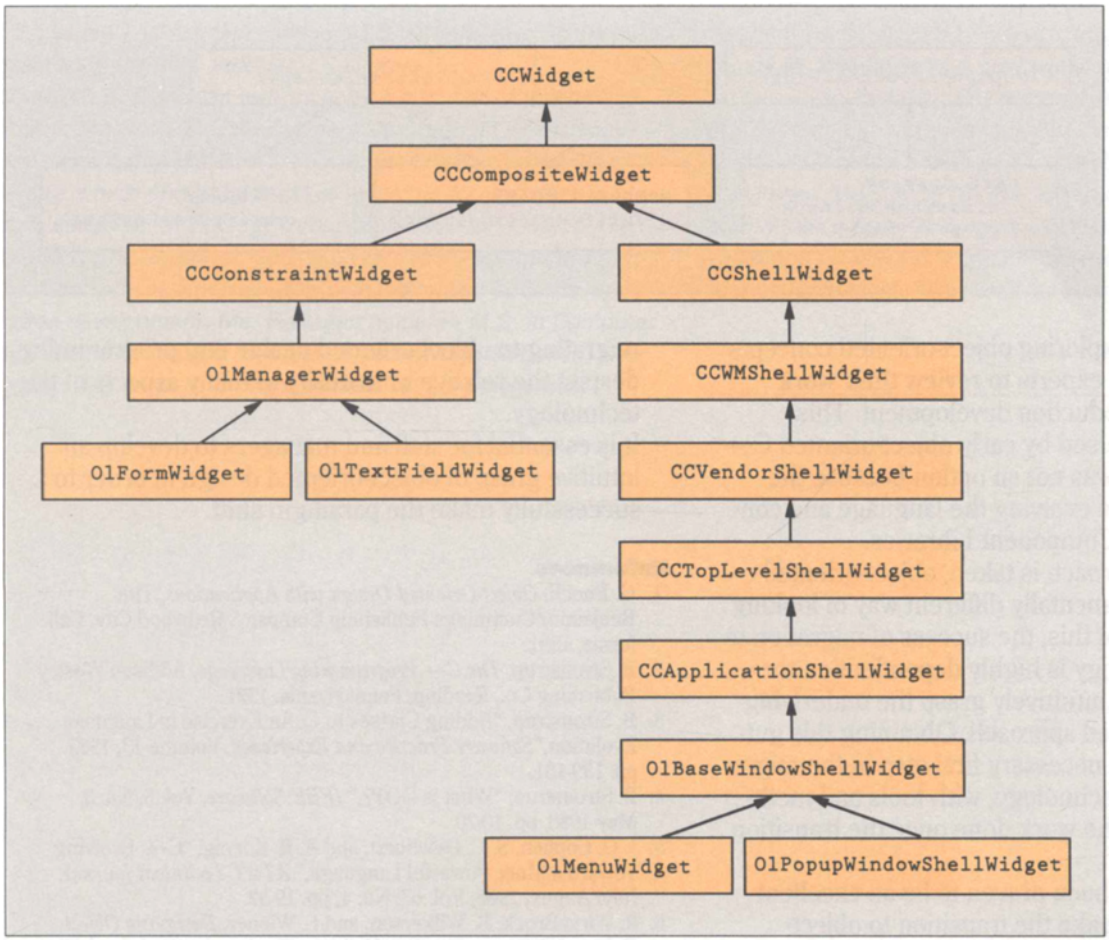

Figure 2. Reusing a class library, such as a Widget library, provides a common "look and feel," and reduces programming complexity. Encapsulation and information hiding of the underlying set protects application code during revisions.

of the C++ class widgets. By encapsulating and hiding the details of the underlying widget set, application code is protected from the not infrequent changes made to the OPEN LOOK Widgets. This library actually forms a framework of classes which can be fleshed out to form a user interface for a particular application.

The example in Figure 3 shows classes from the Widgets library being reused and extended with new member functions which deal with class names, responsibilities and collaborations.

Migrating to Object-Oriented Design and C++

There are many ways a project can migrate to object-oriented technology. All involve some level of training, learning by experience, and input from experts. The amount of each of these varies based on the project goal with respect to speed of migration. Two modes of migration have been the most popular within AT&T: they are generally referred to as the *seeding* and *sandbox* approaches.

Seeding is preferred when projects need to initiate their object-oriented new product development very quickly. This strategy relies heavily on importing expertise. Experts are recruited either as consultants or regular staff. While leading the object-oriented design activities, these experts also work at bringing other project members up to speed. Training and learning-through-experience is still important but the focus on speed demands imported expertise.

A more gradual approach, used by projects who can afford the time, is the sandbox or learning-through-experience approach. After some initial training, staff

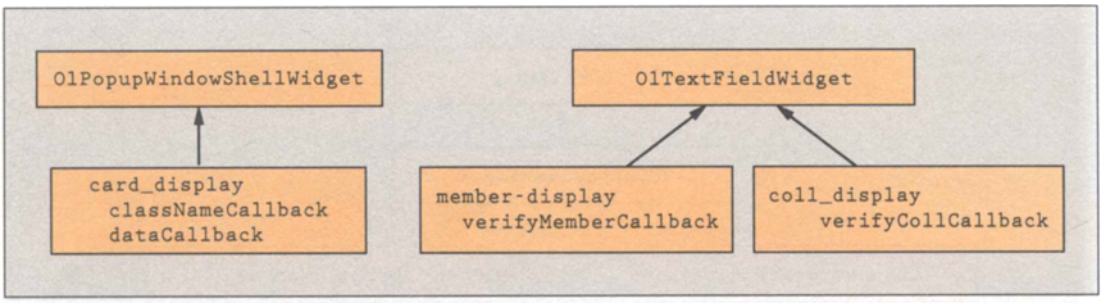

Figure 3. Classes from a library, such as a Widget library, can be reused and extended with new functionality.

members spend time exploring object-oriented concepts and C++, relying on the experts to review their work prior to moving into production development. This approach was typically used by early object-oriented C++ projects when seeding was not an option because the experts were engaged in evolving the language and constructing the Standard Component Libraries.

Whichever approach is taken, object-oriented design requires a fundamentally different way of looking at a problem. Because of this, the success of migration to object-oriented technology is highly dependent on the degree to which people intuitively grasp the underlying idea of the object-oriented approach. Obtaining this gut-level understanding is a necessary first step in the migration to object-oriented technology, with tools and methods simply supporting the work done once the transition is made.

CRC cards have been proven to be an excellent aid in helping projects make the transition to object-oriented technology[6,7,9] as well as for developing a high level system design. The CRC design process helps to identify the classes of the system and the subsystems these classes may form. It also helps to model the class relationships based on the responsibilities of each class and their needs to collaborate or work with one another.

Conclusion

In the future, we expect to see more AT&T projects start in or migrate to object-oriented technology and C++. We also expect to see construction of application domain libraries and the evolution of application frameworks.

In closing, we would like to leave the reader with two key ideas:

- It has been our experience over the last five years that the benefits more than justify the investment in migrating to object-oriented design and programming, despite the relative immaturity of many aspects of the technology.
- It is essential for staff and managers to develop an intuitive grasp of object-oriented design in order to successfully make the paradigm shift.

References

1. G. Booch, *Object-Oriented Design with Applications,* The Benjamin/Cummings Publishing Company, Redwood City, California, 1991.
2. B. Stroustrup, *The C++ Programming Language,* Addison-Wesley Publishing Co., Reading, Pennsylvania, 1991.
3. B. Stroustrup, "Adding Classes to C: An Exercise in Language Evolution,"*Software Practice and Experience*, Volume 13, 1983, pp. 139-161.
4. B. Stroustrup, "What is OOP?," *IEEE Software*, Vol. 5, No. 3. May 1988, pp. 10-20.
5. J. O. Coplien, S. C. Dewhurst, and A. R. Koenig, "C++: Evolving Toward a More Powerful Language," *AT&T Technical Journal*, July/August 1988, Vol. 67, No. 4, pp. 19-32.
6. R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software,* Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1990.
7. Beck, K., "Think Like an Object," *UNIX Review*, Vol. 9, No. 10, pp. 39-43.
8. S. Schlaer and S. Mellor, *Object-Oriented Systems Analysis: Modeling the World in Data,* Yourdon Press, Englewood Cliffs, New Jersey, 1988.
9. J. O. Coplien, "Experience with CRC Cards in AT&T," *The C++ Report*, Vol. 3, No. 8, pp. 1-6.

(Manuscript received July 31, 1992)

Cecilia M. Castillo *is a member of technical staff in the Software Development Environment Technology Department of AT&T Bell Laboratories, Liberty Corner, New Jersey. She is working on object-oriented design. She joined AT&T in 1986.*

from UCLA, Los Angeles, and an M.S. in Computer Science, Stanford University, Palo Alto, California.

Elizabeth B. Flanagan *was recently a member of technical staff in the Software Development Environment Technology Department of AT&T Bell Laboratories, Liberty Corner, New Jersey, where she led a team responsible for transferring object-oriented technology throughout AT&T. She joined AT&T in 1981. She is now at HBO, a division of Time-Warner Entertainment, where she manages object-oriented software application development. Ms. Flanagan holds an M.S. in Computer Science from Stevens Institute of Technology, Hoboken, New Jersey, and an M.S. in Management of Technology from National Technological University, Boulder, Colorado.*

Nancy M. Wilkinson *is a member of technical staff in the Software Development Environment Technology Department in AT&T Bell Laboratories, Liberty Corner, New Jersey. She works on libraries and tools for object-oriented development. She joined AT&T in 1979. Ms. Wilkinson holds a B.S. in Mathematics from Gettysburg College, Gettysburg, Pennsylvania and an M.S. in Computer Science from Stevens Institute of Technology, Hoboken, New Jersey.*